作业三

2215408108 软件工程专业 程乐怡

15-1

1. 问题描述与子问题划分

子问题定义：设$dp[v]$表示从起点s到顶点v的最长加权路径的长度。

状态转移方程：

- 初始条件： dp[s] = 0 （从 s 到自身路径长度为零）。
- 转移条件：对于每个顶点 v，如果存在从 u 到 v 的边，权重为 w(u, v)，则：

$$dp[v] = \max_{(u,v)\in E}\{dp[u] + w(u,v)\}$$

目标是希望求出 dp[t]，即从 s 到 t 的最长路径长度。

2. 算法描述

- **拓扑排序**：对图 G 进行拓扑排序，将顶点按拓扑序排列为 $v_1, v_2, \ldots, v_n$。拓扑排序的时间复杂度为 O(V+E)。
- **动态规划**：按照拓扑顺序遍历顶点，对每个顶点 v，按照状态转移方程更新 dp[v]。对于每条边 (u, v)，执行以下步骤：

  $$dp[v] = \max(dp[v], dp[u] + w(u,v))$$

  这一部分的时间复杂度为 O(E)。
- **返回结果**：最终返回 dp[t]。

3. 子问题图的结构

子问题图是基于拓扑排序构造的一个**有向无环图（DAG）**，其结构与原图相似，但顶点的更新顺序是按照拓扑序执行的。通过动态规划，子问题之间形成了依赖关系，具体表现为每个顶点的值 dp[v] 依赖于其所有前驱节点的值 dp[u] 和边权重 w(u, v)。

4. 算法效率分析

- 拓扑排序时间复杂度：使用 Kahn 算法或 DFS 实现，时间复杂度为 O(V+E)。
- 动态规划时间复杂度：对于每个顶点 v，我们需要检查其所有的入边，时间复杂度为 O(E)。

总时间复杂度为$O(V+E)$，效率较高，适用于 DAG 中最长路径问题。

15-2

1. 子问题定义

设 dp[i][j] 表示字符串 $S[i \ldots j]$ 的最长回文子序列的长度，其中 i 和 j 是字符串的左右边界索引。

2. 状态转移方程

如果 S[i] == S[j]，当字符串两端字符相同时，它们一定属于最长回文子序列，因此：

$$dp[i][j] = dp[i+1][j-1] + 2$$

将 S[i] 和 S[j] 加入子序列

如果 $S[i] \neq S[j]$ ：此时最长回文子序列只能出现在两个子区间之一：

- $S[i+1 \ldots j]$（排除左端字符 S[i]）；
- $S[i \ldots j-1]$（排除右端字符 S[j]）。

$$dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$$

3. 边界条件

当 i == j 时，子字符串仅包含一个字符，显然是回文：

$$dp[i][i] = 1$$

目标是求 dp[0][n-1]，即整个字符串的最长回文子序列的长度。

4. 算法步骤

创建一个二维数组 dp 大小为 $n \times n$，其中 n 是字符串长度。将对角线元素 dp[i][i] = 1 初始化为 1。

按照子问题规模从小到大的顺序（即子串长度从 2 到 n）填充 dp 表，更新每个 dp[i][j] 的值。

根据 dp 表记录的状态，回溯构造出最长回文子序列。

5. 时间与空间复杂度

时间复杂度： dp 表需要填充 $n \times n$ 个状态，每个状态的计算复杂度为 O(1)，总时间复杂度为：O(n^2)

空间复杂度：由于需要存储 dp 表，空间复杂度为：O(n^2)

如果只需要长度，可以通过滚动数组优化空间复杂度为 O(n)。

15-4

给定一段由 n 个单词组成的文本，每个单词长度为$l_1, l_2, \ldots, l_n$，需要将这些单词排版为多行，每行最多容纳 M 个字符（包括单词和单词间的空格），目标是通过动态规划最小化所有行（不包括最后一行）结尾的多余空格数量的立方和。

1. 子问题定义

设 dp[j] 表示将第 1 到第 j 个单词排版所需的最小“代价”（即多余空格立方和的总和）。为了确定状态转移关系，需要考虑从第 i 个单词开始到第 j 个单词结束的代价。

2. 关键变量定义

- **代价函数**：如果第 i 到第 j 个单词被排在同一行，且多余空格数为：

$$extra[i][j] = M - (j - i) - \sum_{k=i}^{j} l_k$$

（其中 (j - i) 表示 j-i 个空格）。

- 如果 $extra[i][j] \geq 0$，则行的代价为：

$$cost[i][j] = extra[i][j]^3$$

- 如果 extra[i][j] < 0，则单词无法排在同一行，此时 $cost[i][j] = \infty$。
- **状态转移方程**：dp[j] 的最优解可以从某个 i 划分过来，即：

（其中 dp[i-1] 表示第 1 到第 i-1 个单词的最优代价）。

$$dp[j] = \min_{1 \leq i \leq j} \{dp[i-1] + cost[i][j]\}$$

- **目标**：最小化从第 1 到第 n 个单词的代价，即求 dp[n]。

3. 算法步骤

- **预计算空格和代价表**：
  - 计算所有可能的 extra[i][j] 值；
  - 根据 extra[i][j] 的正负，构建 cost[i][j] 表。
- **动态规划填表**：
  - 初始化：dp[0] = 0 （表示没有单词的代价为 0）；
  - 对于 $j = 1, 2, \ldots, n$，依次计算 dp[j]：

$$dp[j] = \min_{1 \leq i \leq j} \{dp[i-1] + cost[i][j]\}$$

- **回溯输出排版方案**，从 dp 表中回溯，找到每一行的起止单词索引。

4. 时间和空间复杂度

时间复杂度：预计算 extra[i][j] 和 cost[i][j] 需要 O(n^2)；填充 dp[j] 表需要 O(n^2)，因此总时间复杂度为 O(n^2)

空间复杂度：extra[i][j] 和 cost[i][j] 占用 O(n^2) 空间；dp[j] 占用 O(n) 空间；总空间复杂度为O(n^2)

```
Input: n words with lengths l[1..n], maximum line width M
Output: Minimum cost dp[n] and optimal line breaks

1. Precompute extra and cost tables:
   Initialize extra[i][j] and cost[i][j] for all 1 ≤ i ≤ j ≤ n:
   for i = 1 to n:
       extra[i][i] = M - l[i]
       for j = i+1 to n:
```

```
            extra[i][j] = extra[i][j-1] - l[j] - 1

    for i = 1 to n:
        for j = i to n:
            if extra[i][j] < 0:
                cost[i][j] = ∞
            else if j == n:
                cost[i][j] = 0
            else:
                cost[i][j] = extra[i][j]^3

2. Compute the dp table:
    Initialize dp[0] = 0
    for j = 1 to n:
        dp[j] = ∞
        for i = 1 to j:
            if dp[i-1] + cost[i][j] < dp[j]:
                dp[j] = dp[i-1] + cost[i][j]
                line_break[j] = i

3. Reconstruct the solution:
    Initialize k = n
    lines = []
    while k > 0:
        i = line_break[k]
        lines.append(words[i..k])
        k = i - 1

4. Return dp[n] (minimum cost) and lines (formatted text)
```